

# I never understood JavaScript closures

*Translation: 我从不理解JavaScript闭包*

**Until someone explained it to me like this ...**

*Translation: 直到有人这样向我解释...*

As the title states, JavaScript closures have always been a bit of a mystery to me. I have read multiple articles, I have used closures in my work, sometimes I even used a closure without realizing I was using a closure.

*Translation: 正如标题所说, JavaScript闭包对我来说一直有点神秘。我读过多篇文章, 我在工作中使用过闭包, 有时我甚至在没有意识到自己在使用闭包的情况下使用了闭包。*

Recently I went to a talk where someone really explained it in a way it finally clicked for me. I'll try to take this approach to explain closures in this article. Let me give credit to the great folks at CodeSmith and their JavaScript The Hard Parts series.

*Translation: 最近, 我参加了一个演讲, 有人真的用一种我终于接受的方式解释了它。我将在本文中尝试用这种方法来解释闭包。让我向CodeSmith和他们的JavaScript the Hard Parts系列的伟大员工致敬。*

## Before we start

Some concepts are important to grok before you can grok closures. One of them is the execution context.

*Translation: 在您可以grok闭包之前, 有些概念对grok很重要。其中之一是执行上下文。*

This article has a very good primer on Execution Context. To quote the article:

*Translation: 这篇文章对执行上下文有很好的入门知识。引用这篇文章:*

When code is run in JavaScript, the environment in which it is executed is very important, and is evaluated as 1 of the following:

*Translation: 当代码在JavaScript中运行时, 执行代码的环境非常重要, 评估为以下之一:*

Global code — The default environment where your code is executed for the first time.

Function code — Whenever the flow of execution enters a function body.

(...)

(...), let's think of the term execution context as the environment / scope the current code is being evaluated in.

*Translation: (...), 让我们将术语执行上下文视为当前代码正在评估的环境/范围。*

In other words, as we start the program, we start in the global execution context. Some variables are declared within the global execution context. We call these global variables. When the program calls a function, what happens? A few steps:

*Translation: 换句话说, 当我们启动程序时, 我们从全局执行上下文中开始。一些变量是在全局执行上下文中声明的。我们称这些为全局变量。当程序调用一个函数时, 会发生什么? 几个步骤:*

JavaScript creates a new execution context, a local execution context

That local execution context will have its own set of variables, these variables will be local to that execution context.

The new execution context is thrown onto the execution stack. Think of the execution stack as a mechanism to keep track of where the program is in its execution

When does the function end? When it encounters a return statement or it encounters a closing bracket `}`. When a function ends, the following happens:

*Translation: 功能何时结束? 当它遇到返回语句或遇到右括号时`}`。当函数结束时, 会发生以下情况:*

The local execution contexts pops off the execution stack

*Translation: 本地执行上下文从执行堆栈中弹出*

The functions sends the return value back to the calling context. The calling context is the execution context that called this function, it could be the global execution context or another local execution context. It is up to the calling execution context to deal with the return value at that point. The returned value could be an object, an array, a function, a boolean, anything really. If the function has no return statement, undefined is returned.

*Translation: 函数将返回值发送回调用上下文。调用上下文是调用此函数的执行上下文, 它可以是全局执行上下文或另一个本地执行上下文。这取决于调用执行上下文来处理此时的*

返回值。返回的值可以是对象、数组、函数、布尔值，也可以是任何实际值。如果函数没有返回语句，则返回undefined。

The local execution context is destroyed. This is important. Destroyed. All the variables that were declared within the local execution context are erased. They are no longer available. That's why they're called local variables.

## A very basic example

Before we get to closures, let's take a look at the following piece of code. It seems very straightforward, anybody reading this article probably knows exactly what it does.

```
1: let a = 3
2: function addTwo(x) {
3:   let ret = x + 2
4:   return ret
5: }
6: let b = addTwo(a)
7: console.log(b)
```

*Translation: 1: 设a=3*

In order to understand how the JavaScript engine really works, let's break this down in great detail.

*Translation: 为了了解JavaScript引擎是如何真正工作的，让我们详细分析一下。*

On line 1 we declare a new variable a in the global execution context and assign it the number 3.

Next it gets tricky. Lines 2 through 5 are really together. What happens here? We declare a new variable named addTwo in the global execution context. And what do we assign to it? A function definition. Whatever is between the two brackets { } is assigned to addTwo. The code inside the function is not evaluated, not executed, just stored into a variable for future use.

So now we're at line 6. It looks simple, but there is much to unpack here. First we declare a new variable in the global execution context and label it b. As soon as a variable is declared it has the value of undefined.

Next, still on line 6, we see an assignment operator. We are getting ready to assign a new value to the variable b. Next we see a function being called. When you see a variable followed by round brackets (...), that's the signal that a function is being called. Flash forward, every function returns something (either a value, an object or undefined). Whatever is returned from the function will be assigned to variable b.

But first we need to call the function labeled `addTwo`. JavaScript will go and look in its global execution context memory for a variable named `addTwo`. Oh, it found one, it was defined in step 2 (or lines 2–5). And lo and behold variable `addTwo` contains a function definition. Note that the variable `a` is passed as an argument to the function. JavaScript searches for a variable `a` in its global execution context memory, finds it, finds that its value is 3 and passes the number 3 as an argument to the function. Ready to execute the function.

*Translation: 但首先我们需要调用标记为addTwo的函数。JavaScript将在其全局执行上下文内存中查找名为addTwo的变量。哦，它找到了一个，它是在第2步（或第2-5行）中定义的。瞧，变量addTwo包含一个函数定义。请注意，变量a是作为参数传递给函数的。JavaScript在其全局执行上下文内存中搜索变量a，找到它，发现它的值为3，并将数字3作为参数传递给函数。准备执行功能。*

Now the execution context will switch. A new local execution context is created, let's name it the 'addTwo execution context'. The execution context is pushed onto the call stack. What is the first thing we do in the local execution context?

You may be tempted to say, "A new variable `ret` is declared in the local execution context". That is not the answer. The correct answer is, we need to look at the parameters of the function first. A new variable `x` is declared in the local execution context. And since the value 3 was passed as an argument, the variable `x` is assigned the number 3.

The next step is: A new variable `ret` is declared in the local execution context. Its value is set to undefined. (line 3)

Still line 3, an addition needs to be performed. First we need the value of `x`. JavaScript will look for a variable `x`. It will look in the local execution context first. And it found one, the value is 3. And the second operand is the number 2. The result of the addition (5) is assigned to the variable `ret`.

Line 4. We return the content of the variable `ret`. Another lookup in the local execution context. `ret` contains the value 5. The function returns the number 5. And the function ends.

*Translation: 第4行。我们返回变量ret的内容。在本地执行上下文中进行另一次查找。ret包含值5。函数返回数字5。功能结束。*

Lines 4–5. The function ends. The local execution context is destroyed. The variables `x` and `ret` are wiped out. They no longer exist. The context is popped of the call stack and the return value is returned to the calling context. In this case the calling context is the global execution context, because the function `addTwo` was called from the global execution context.

*Translation: 第4–5行。函数结束。本地执行上下文已销毁。变量x和ret被擦除。它们已不复存在。从调用堆栈中弹出上下文，并将返回值返回给调用上下文。在这种情况下，调用上下文是全局执行上下文，因为函数addTwo是从全局执行上下文调用的。*

Now we pick up where we left off in step 4. The returned value (number 5) gets assigned to the variable b. We are still at line 6 of the little program.

I am not going into detail, but in line 7, the content of variable b gets printed in the console. In our example the number 5.

That was a very long winded explanation for a very simple program, and we haven't even touched upon closures yet. We will get there I promise. But first we need to take another detour or two.

## Lexical scope.

We need to understand some aspects of lexical scope. Take a look at the following example.

*Translation: 我们需要了解词汇范围的某些方面。看看下面的例子。*

```
1: let val1 = 2
2: function multiplyThis(n) {
3:   let ret = n * val1
4:   return ret
5: }
6: let multiplied = multiplyThis(6)
7: console.log('example of scope:', multiplied)
```

*Translation: 1: 设val1=2*

The idea here is that we have variables in the local execution context and variables in the global execution context. One intricacy of JavaScript is how it looks for variables. If it can't find a variable in its local execution context, it will look for it in its calling context. And if not found there in its calling context. Repeatedly, until it is looking in the global execution context. (And if it does not find it there, it's undefined). Follow along with the example above, it will clarify it. If you understand how scope works, you can skip this.

Declare a new variable val1 in the global execution context and assign it the number 2.

Lines 2–5. Declare a new variable multiplyThis and assign it a function definition.

Line 6. Declare a new variable multiplied in the global execution context.

Retrieve the variable multiplyThis from the global execution context memory and execute it as a function. Pass the number 6 as argument.

*Translation: 从全局执行上下文内存中检索变量multiplyThis，并将其作为函数执行。将数字6作为参数传递。*

New function call = new execution context. Create a new local execution context.

*Translation: 新函数调用=新的执行上下文。创建一个新的本地执行上下文。*

In the local execution context, declare a variable `n` and assign it the number 6.

Line 3. In the local execution context, declare a variable `ret`.

Line 3 (continued). Perform an multiplication with two operands; the content of the variables `n` and `val1`. Look up the variable `n` in the local execution context. We declared it in step 6. Its content is the number 6. Look up the variable `val1` in the local execution context. The local execution context does not have a variable labeled `val1`. Let's check the calling context. The calling context is the global execution context. Let's look for `val1` in the global execution context. Oh yes, it's there. It was defined in step 1. The value is the number 2.

Line 3 (continued). Multiply the two operands and assign it to the `ret` variable.  $6 * 2 = 12$ . `ret` is now 12.

*Translation: 第3行 (续)。将两个操作数相乘，并将其分配给ret变量。6 \* 2 = 12. 雷特现在12岁了。*

Return the `ret` variable. The local execution context is destroyed, along with its variables `ret` and `n`. The variable `val1` is not destroyed, as it was part of the global execution context.

*Translation: 返回ret变量。本地执行上下文及其变量ret和n被销毁。变量val1没有被销毁，因为它是全局执行上下文的一部分。*

Back to line 6. In the calling context, the number 12 is assigned to the multiplied variable.

Finally on line 7, we show the value of the multiplied variable in the console.

So in this example, we need to remember that a function has access to variables that are defined in its calling context. The formal name of this phenomenon is the lexical scope.

## **A function that returns a function**

*Translation: 返回函数的函数*

In the first example the function `addTwo` returns a number. Remember from earlier that a function can return anything. Let's look at an example of a function that returns a function, as this is essential to understand closures. Here is the example that we are going to analyze.

*Translation: 在第一个例子中，函数addTwo返回一个数字。请记住前面提到的函数可以返回任何内容。让我们看一个返回函数的函数示例，因为这对于理解闭包至关重要。这是我们要分析的例子。*

```
1: let val = 7
2: function createAdder() {
3:   function addNumbers(a, b) {
4:     let ret = a + b
5:     return ret
6:   }
7:   return addNumbers
8: }
9: let adder = createAdder()
10: let sum = adder(val, 8)
11: console.log('example of function returning a function: ', sum)
```

Let's go back to the step-by-step breakdown.

Line 1. We declare a variable `val` in the global execution context and assign the number 7 to that variable.

Lines 2–8. We declare a variable named `createAdder` in the global execution context and we assign a function definition to it. Lines 3 to 7 describe said function definition. As before, at this point, we are not jumping into that function. We just store the function definition into that variable (`createAdder`).

Line 9. We declare a new variable, named `adder`, in the global execution context. Temporarily, `undefined` is assigned to `adder`.

*Translation: 第9行。我们在全局执行上下文中声明一个新的变量，名为adder。暂时将未定义分配给加法器。*

Still line 9. We see the brackets `()`; we need to execute or call a function. Let's query the global execution context's memory and look for a variable named `createAdder`. It was created in step 2. Ok, let's call it.

Calling a function. Now we're at line 2. A new local execution context is created. We can create local variables in the new execution context. The engine adds the new context to the call stack. The function has no arguments, let's jump right into the body of it.

Still lines 3–6. We have a new function declaration. We create a variable `addNumbers` in the local execution context. This is important. `addNumbers` exists only in the local execution context. We store a function definition in the local variable named `addNumbers`.

Now we're at line 7. We return the content of the variable `addNumbers`. The engine looks for a variable named `addNumbers` and finds it. It's a function definition. Fine, a function can return

anything, including a function definition. So we return the definition of addNumbers. Anything between the brackets on lines 4 and 5 makes up the function definition. We also remove the local execution context from the call stack.

Upon return, the local execution context is destroyed. The addNumbers variable is no more. The function definition still exists though, it is returned from the function and it is assigned to the variable adder; that is the variable we created in step 3.

Now we're at line 10. We define a new variable sum in the global execution context. Temporary assignment is undefined.

*Translation: 现在我们在10号线。我们在全局执行上下文中定义了一个新的变量和。临时分配未定义。*

We need to execute a function next. Which function? The function that is defined in the variable named adder. We look it up in the global execution context, and sure enough we find it. It's a function that takes two parameters.

*Translation: 接下来我们需要执行一个函数。哪个功能？在名为adder的变量中定义的函数。我们在全局执行上下文中查找它，肯定会找到它。它是一个需要两个参数的函数。*

Let's retrieve the two parameters, so we can call the function and pass the correct arguments. The first one is the variable val, which we defined in step 1, it represents the number 7, and the second one is the number 8.

Now we have to execute that function. The function definition is outlined lines 3–5. A new local execution context is created. Within the local context two new variables are created: a and b. They are respectively assigned the values 7 and 8, as those were the arguments we passed to the function in the previous step.

Line 4. A new variable is declared, named ret. It is declared in the local execution context.

Line 4. An addition is performed, where we add the content of variable a and the content of variable b. The result of the addition (15) is assigned to the ret variable.

The ret variable is returned from that function. The local execution context is destroyed, it is removed from the call stack, the variables a, b and ret no longer exist.

*Translation: ret变量是从该函数返回的。本地执行上下文被破坏，它被从调用堆栈中删除，变量a、b和ret不再存在。*

The returned value is assigned to the sum variable we defined in step 9.

We print out the value of sum to the console.



As expected the console will print 15. We really go through a bunch of hoops here. I am trying to illustrate a few points here. First, a function definition can be stored in a variable, the function definition is invisible to the program until it gets called. Second, every time a function gets called, a local execution context is (temporarily) created. That execution context vanishes when the function is done. A function is done when it encounters return or the closing bracket }.

## Finally, a closure

Take a look at the next code and try to figure out what will happen.

```
1: function createCounter() {
2:   let counter = 0
3:   const myFunction = function() {
4:     counter = counter + 1
5:     return counter
6:   }
7:   return myFunction
8: }
9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

*Translation: 1:函数createCounter () {*

Now that we got the hang of it from the previous two examples, let's zip through the execution of this, as we expect it to run.

*Translation: 既然我们已经从前两个例子中了解了它的窍门，让我们快速完成它的执行，因为我们希望它能运行。*

Lines 1–8. We create a new variable createCounter in the global execution context and it gets assigned function definition.

Line 9. We declare a new variable named increment in the global execution context..

Line 9 again. We need call the createCounter function and assign its returned value to the increment variable.

Lines 1–8 . Calling the function. Creating new local execution context.

*Translation: 第1-8行。正在调用函数。正在创建新的本地执行上下文。*

Line 2. Within the local execution context, declare a new variable named counter. Number 0 is assigned to counter.

*Translation: 第2行。在本地执行上下文中，声明一个名为counter的新变量。编号0分配给计数器。*

Line 3–6. Declaring new variable named myFunction. The variable is declared in the local execution context. The content of the variable is yet another function definition. As defined in lines 4 and 5.

Line 7. Returning the content of the myFunction variable. Local execution context is deleted. myFunction and counter no longer exist. Control is returned to the calling context.

Line 9. In the calling context, the global execution context, the value returned by createCounter is assigned to increment. The variable increment now contains a function definition. The function definition that was returned by createCounter. It is no longer labeled myFunction, but it is the same definition. Within the global context, it is labeled increment.

Line 10. Declare a new variable (c1).

Line 10 (continued). Look up the variable increment, it's a function, call it. It contains the function definition returned from earlier, as defined in lines 4–5.

*Translation: 第10行 (续)。查找变量increment，它是一个函数，调用它。它包含从前面返回的函数定义，如第4-5行所定义。*

Create a new execution context. There are no parameters. Start execution the function.

*Translation: 创建一个新的执行上下文。没有参数。开始执行函数。*

Line 4. counter = counter + 1. Look up the value counter in the local execution context. We just created that context and never declare any local variables. Let's look in the global execution context. No variable labeled counter here. Javascript will evaluate this as counter = undefined + 1, declare a new local variable labeled counter and assign it the number 1, as undefined is sort of 0.

Line 5. We return the content of counter, or the number 1. We destroy the local execution context, and the counter variable.

Back to line 10. The returned value (1) gets assigned to c1.

Line 11. We repeat steps 10–14, c2 gets assigned 1 also.

*Translation: 第11行。我们重复步骤10-14，c2也被赋值为1。*

Line 12. We repeat steps 10–14, c3 gets assigned 1 also.

*Translation: 第12行。我们重复步骤10-14，c3也被赋值为1。*

Line 13. We log the content of variables c1, c2 and c3.

*Translation: 第13行。我们记录变量c1、c2和c3的内容。*

Try this out for yourself and see what happens. You'll notice that it is not logging 1, 1, and 1 as you may expect from my explanation above. Instead it is logging 1, 2 and 3. So what gives?

Somehow, the increment function remembers that counter value. How is that working?

Is counter part of the global execution context? Try `console.log(counter)` and you'll get undefined. So that's not it.

*Translation: 计数器是全局执行上下文的一部分吗？试试`console.log (counter)`，你会得到未定义的结果。所以不是这样。*

Maybe, when you call increment, somehow it goes back to the the function where it was created (createCounter)? How would that even work? The variable increment contains the function definition, not where it came from. So that's not it.

*Translation: 也许，当您调用increment时，它会以某种方式返回到创建它的函数 (createCounter)？这是怎么回事？变量增量包含函数定义，而不是它的来源。所以不是这样。*

So there must be another mechanism. The Closure. We finally got to it, the missing piece.

*Translation: 因此，必须有另一种机制。结束。我们终于找到了它，丢失的那一块。*

Here is how it works. Whenever you declare a new function and assign it to a variable, you store the function definition, as well as a closure. The closure contains all the variables that are in scope at the time of creation of the function. It is analogous to a backpack. A function definition comes with a little backpack. And in its pack it stores all the variables that were in scope at the time that the function definition was created.

So our explanation above was all wrong, let's try it again, but correctly this time.

```
1: function createCounter() {  
  2:   let counter = 0  
  3:   const myFunction = function() {  
  4:     counter = counter + 1  
  5:     return counter  
  6:   }
```

```
7:   return myFunction
8: }
9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

Lines 1–8. We create a new variable `createCounter` in the global execution context and it gets assigned function definition. Same as above.

*Translation: 第1-8行。我们在全局执行上下文中创建了一个新的变量`createCounter`，它得到了指定的函数定义。同上。*

Line 9. We declare a new variable named `increment` in the global execution context. Same as above.

*Translation: 第9行。我们在全局执行上下文中声明一个名为`increment`的新变量。同上。*

Line 9 again. We need call the `createCounter` function and assign its returned value to the `increment` variable. Same as above.

*Translation: 第9行。我们需要调用`createCounter`函数，并将其返回值分配给增量变量。同上。*

Lines 1–8 . Calling the function. Creating new local execution context. Same as above.

Line 2. Within the local execution context, declare a new variable named `counter`. Number 0 is assigned to `counter`. Same as above.

Line 3–6. Declaring new variable named `myFunction`. The variable is declared in the local execution context. The content of the variable is yet another function definition. As defined in lines 4 and 5. Now we also create a closure and include it as part of the function definition. The closure contains the variables that are in scope, in this case the variable `counter` (with the value of 0).

*Translation: 第3-6行。正在声明名为`myFunction`的新变量。变量是在本地执行上下文中声明的。变量的内容是另一个函数定义。如第4行和第5行所定义。现在，我们还创建了一个闭包，并将其作为函数定义的一部分。闭包包含作用域中的变量，在本例中为变量计数器（值为0）。*

Line 7. Returning the content of the `myFunction` variable. Local execution context is deleted. `myFunction` and `counter` no longer exist. Control is returned to the calling context. So we are returning the function definition and its closure, the backpack with the variables that were in scope when it was created.

*Translation: 第7行。返回myFunction变量的内容。本地执行上下文已删除。myFunction和counter已不存在。控件返回到调用上下文。因此，我们将返回函数定义及其闭包，即背包，其中包含创建时在范围内的变量。*

Line 9. In the calling context, the global execution context, the value returned by createCounter is assigned to increment. The variable increment now contains a function definition (and closure). The function definition that was returned by createCounter. It is no longer labeled myFunction, but it is the same definition. Within the global context, it is called increment.

*Translation: 第9行。在调用上下文，即全局执行上下文中，createCounter返回的值被分配给increment。变量increment现在包含一个函数定义（和闭包）。createCounter返回的函数定义。它不再被标记为myFunction，但它是相同的定义。在全球范围内，它被称为增量。*

Line 10. Declare a new variable (c1).

*Translation: 第10行。声明一个新变量 (c1) 。*

Line 10 (continued). Look up the variable increment, it's a function, call it. It contains the function definition returned from earlier, as defined in lines 4–5. (and it also has a backpack with variables)

*Translation: 第10行（续）。查找变量increment，它是一个函数，调用它。它包含从前面返回的函数定义，如第4-5行所定义。（它还有一个带有变量的背包）*

Create a new execution context. There are no parameters. Start execution the function.

*Translation: 创建一个新的执行上下文。没有参数。开始执行函数。*

Line 4. counter = counter + 1. We need to look for the variable counter. Before we look in the local or global execution context, let's look in our backpack. Let's check the closure. Lo and behold, the closure contains a variable named counter, its value is 0. After the expression on line 4, its value is set to 1. And it is stored in the backpack again. The closure now contains the variable counter with a value of 1.

*Translation: 第4行。counter=counter+1。我们需要查找变量计数器。在我们研究本地或全局执行上下文之前，让我们先看看我们的背包。让我们检查一下闭合情况。瞧，闭包包含一个名为counter的变量，其值为0。在第4行的表达式之后，其值设置为1。然后它又被存放在背包里。闭包现在包含值为1的变量计数器。*

Line 5. We return the content of counter, or the number 1. We destroy the local execution context.

*Translation: 第5行。我们返回计数器的内容，或者数字1。我们破坏了本地执行上下文。*

Back to line 10. The returned value (1) gets assigned to c1.

*Translation: 回到第10行。返回的值 (1) 被分配给c1。*

Line 11. We repeat steps 10–14. This time, when we look at our closure, we see that the counter variable has a value of 1. It was set in step 12 or line 4 of the program. Its value gets incremented and stored as 2 in the closure of the increment function. And c2 gets assigned 2.

*Translation: 第11行。我们重复步骤10-14。这一次，当我们查看闭包时，我们看到计数器变量的值为1。它是在程序的第12步或第4行中设置的。它的值被递增，并存储为递增函数的闭包中的2。c2得到2。*

Line 12. We repeat steps 10–14, c3 gets assigned 3.

*Translation: 第12行。我们重复步骤10-14，c3被分配为3。*

Line 13. We log the content of variables c1, c2 and c3.

*Translation: 第13行。我们记录变量c1、c2和c3的内容。*

So now we understand how this works. The key to remember is that when a function gets declared, it contains a function definition and a closure. The closure is a collection of all the variables in scope at the time of creation of the function.

*Translation: 现在我们明白了这是怎么回事。需要记住的关键是，当一个函数被声明时，它包含一个函数定义和一个闭包。闭包是在创建函数时作用域中所有变量的集合。*

You may ask, does any function has a closure, even functions created in the global scope? The answer is yes. Functions created in the global scope create a closure. But since these functions were created in the global scope, they have access to all the variables in the global scope. And the closure concept is not really relevant.

*Translation: 您可能会问，是否有任何函数具有闭包，甚至是在全局范围内创建的函数？答案是肯定的。在全局作用域中创建的函数会创建一个闭包。但由于这些函数是在全局范围内创建的，因此它们可以访问全局范围内的所有变量。而闭包概念并不是真正相关的。*

When a function returns a function, that is when the concept of closures becomes more relevant. The returned function has access to variables that are not in the global scope, but they solely exist in its closure.

*Translation: 当一个函数返回一个函数时，也就是闭包的概念变得更加相关的时候。返回的函数可以访问不在全局范围内的变量，但这些变量仅存在于其闭包中。*

## **Not so trivial closures**

*Translation: 不那么琐碎的闭包*

Sometimes closures show up when you don't even notice it. You may have seen an example of what we call partial application. Like in the following code.

*Translation: 有时闭包会在你根本没有注意到的时候出现。你可能已经看到了一个我们称之为部分应用程序的例子。就像下面的代码一样。*

```
let c = 4
const addX = x => n => n + x
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d)
```

*Translation: 设c=4*

In case the arrow function throws you off, here is the equivalent.

*Translation: 如果箭头函数将您抛出，这里是等效的。*

```
let c = 4
function addX(x) {
  return function(n) {
    return n + x
  }
}
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d)
```

*Translation: 设c=4*

We declare a generic adder function addX that takes one parameter (x) and returns another function.

*Translation: 我们声明一个通用的加法器函数addX，它接受一个参数 (x) 并返回另一个函数。*

The returned function also takes one parameter and adds it to the variable x.

*Translation: 返回的函数还接受一个参数并将其添加到变量x中。*

The variable x is part of the closure. When the variable addThree gets declared in the local context, it is assigned a function definition and a closure. The closure contains the variable x.

*Translation: 变量x是闭包的一部分。当变量addThree在本地上下文中声明时，它将被分配一个函数定义和一个闭包。闭包包含变量x。*

So now when addThree is called and executed, it has access to the variable x from its closure and the variable n which was passed as an argument and is able to return the sum.

*Translation: 因此，现在当addThree被调用和执行时，它可以从其闭包访问变量x和作为参数传递的变量n，并能够返回总和。*

In this example the console will print the number 7.

*Translation: 在本例中，控制台将打印数字7。*

## Conclusion

*Translation: 结论*

The way I will always remember closures is through the backpack analogy. When a function gets created and passed around or returned from another function, it carries a backpack with it. And in the backpack are all the variables that were in scope when the function was declared.

*Translation: 我将永远记住封口的方式是通过背包的比喻。当一个函数被创建、传递或从另一个函数返回时，它会携带一个背包。背包中有声明该函数时范围内的所有变量。*

If you enjoyed reading this, don't forget the applause. 🙌 Thank you.

*Translation: 如果你喜欢读这篇文章，别忘了掌声。🙌*